Superfluid Finance GDA

Security Assessment

December 15, 2023

Prepared for:
Miao Zhicheng
Superfluid Finance

Prepared by: **Alexander Remie, Bo Henderson, and Priyanka Bose**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Jeff Braswell, Project Manager
jeff.braswell@trailofbits.com

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Alexander Remie, Consultant
alexander.remie@trailofbits.com

Bo Henderson, Consultant
bo.henderson@trailofbits.com

Priyanka Bose, Consultant
priyanak.bose@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
October 10, 2023	Pre-project kickoff call
October 17, 2023	Status update meeting
November 2, 2023	Report readout meeting; Delivery of report draft
December 1, 2023	Delivery of comprehensive report
December 15, 2023	Delivery of comprehensive report with fix review appendix

Executive Summary

Engagement Overview

Superfluid Finance engaged Trail of Bits to review the security of its extension to the Superfluid protocol called the General Distribution Agreement (GDA). Existing agreements include the Constant Flow Agreement (CFA), which allows users to stream funds to one recipient, and the Instant Distribution Agreement (IDA), which allows users to distribute a payment among many recipients. The GDA provides features from both prior agreements, which allows users to stream funds to many recipients.

A team of three consultants conducted the review from October 11 to October 31, 2023, for a total of four engineer-weeks of effort. Our testing efforts focused on incorrect or missing access controls, liquidations and the solvency mechanism, and the distribution flows. With full access to source code and documentation, we performed static and dynamic testing of the GDA-related contracts, using automated and manual processes. This review was limited to the GDA-related contracts, so we did not review the remainder of the Superfluid protocol.

Observations and Impact

Superfluid's GDA extension uses adequate access controls and is protected against potential attack vectors such as reentrancy and front running. We did not identify any serious issues. However, we did uncover some issues related to bitwise operations (TOB-SUPERFLUID-5) and out-of-gas error reporting (TOB-SUPERFLUID-6) that could become a problem in a future upgrade of the implementation.

Superfluid's test suite is considered very thorough. There are unit tests, integration tests, fuzzing tests, invariant tests (using both Foundry and Echidna), and tests using the Certora Prover. Having such a thorough testing suite helps uncover bugs before deployment. We recommend continuing this practice and extending and improving the test suite as the implementation is updated.

The implementation partially implements the concepts outlined in the Semantic Money paper. These concepts use a functional programming paradigm (Haskell), and the Solidity implementation is written in a functional programming style to mimic the paper. Since the Solidity language is an imperative style language, the current functional style implementation increases the complexity. Consider moving away from this style so that the implementation's size and complexity are decreased.

Inline and NatSpec comments throughout the implementation are very limited. Due to the added complexity of the functional programming style, this is even more of a problem in terms of understanding the implementation. We recommend adding NatSpec and inline comments throughout the implementation and using very verbose, explanatory comments in the files that heavily rely on the functional programming style (`TokenMonad` and `SemanticMoney`).

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Superfluid Finance take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.

- **Improve the inline and NatSpec comments throughout the implementation.** Most of the implementation lacks inline and NatSpec comments. Adding such comments would help readers, auditors, and developers understand the protocol. Some of the contracts are very complex and difficult to understand (`TokenMonad` and `SemanticMoney`).

- **Consider moving away from the functional programming style and using the imperative style instead.** This would decrease the protocol's complexity, which would help developers, auditors, and users understand the implementation and would lower the possibility of bugs slipping through.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	0
Low	0
Informational	6
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Auditing and Logging	2
Data Validation	3
Error Reporting	1

Project Goals

The engagement was scoped to provide a security assessment of the Superfluid Finance GDA–related contracts. Specifically, we sought to answer the following non-exhaustive list of questions:

General

- Are there flaws in the internal accounting of user balances?
- Do all functions have appropriate access controls?
- Is the internal accounting for disconnected members correct?
- Do all configuration and critical functions emit events?
- Does the unsafe upcasting and downcasting of integers lead to problems?
- Is the process of tightly packing variables correctly implemented?
- Are there flaws in the pool creation mechanism?
- Do the various NFTs adhere to the ERC-721 standard?
- Are the flow rates and settled values updated correctly during the distribution flows?
- Does the liquidation mechanism correctly implement the various periods (patrician, plebs, insolvent)?
- Can an attacker steal funds through the distribution or liquidation mechanisms?

Project Targets

The engagement involved a review and testing of the following target.

Superfluid GDA

Repository	https://github.com/superfluid-finance/protocol-monorepo/
Packages	`ethereum-contracts` (only the `GeneralDistributionAgreementV1` and `SuperfluidPool` contracts, and their dependencies), `solidity-semantic-money` (only the `SemanticMoney` and `TokenMonad` contracts, and their dependencies)
Version	`4ece1a3f4aff8b5a9cbf37118d261023960c0f0f`
Type	Solidity
Platform	EVM

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **`GeneralDistributionAgreementV1.sol`**. This contract defines the core business logic governing streamed payments to many recipients. It deploys new pools and manages important internal accounting functionality for each pool. We reviewed the deployment and initialization process for this contract and found no incorrect or error-prone steps. We also reviewed the distribution and liquidation flows for any flaws. The access controls and the internal arithmetic, including its reliance on the `TokenMonad` and `SemanticMoney` contracts, were analyzed for correctness. Finally, we reviewed the encoding and decoding of data into `bytes32` values (TOB-SUPERFLUID-5).

- **`SuperfluidPool.sol`**. Each pool represents one set of many recipients that can receive instant or streamed payments. We reviewed the access controls on the admin role, which can assign units (shares) in the pool's distribution, as well as the internal accounting surrounding connected and disconnected members.

- **`TokenMonad.sol`**. This contract is inherited by the GDA component and provides important arithmetic logic to manage the transfer of pool units and the distribution of value flows. We reviewed the tight integration between the `TokenMonad` contract and the abstract functions implemented by the GDA component. We also reviewed the arithmetic of the various functions and how they use the `SemanticMoney` functions.

- **`SemanticMoney.sol`**. This library defines multiple custom types (e.g., `Time`, `Value`, `Unit`, and `FlowRate`) and structures (e.g., `BasicParticle`) that are heavily used throughout GDA and `SuperfluidPool`. We reviewed the contained arithmetic and how other components use it.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- The GDA is one element of a broader system that includes but is not limited to the Superfluid host, Superfluid tokens, and Superfluid apps. We reviewed the GDA and its dependencies in isolation and reviewed the Superfluid token only to the extent necessary to understand the GDA's functionality. Additional vulnerabilities may be present in the interactions between these components.

- The `TOGA` contract implements the auction mechanism to decide on a Patrician in Charge (PIC) for a given super token. Due to time constraints, we were unable to review this contract. We recommend additional review for this component.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tool in the automated testing phase of this project:

Tool	Description	Policy
Slither	A static analysis framework that can statically verify algebraic relationships between Solidity variables	N/A

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	A modern Solidity compiler version and zero unchecked blocks means no values silently overflow or underflow. Remainders resulting from precision-losing computations are handled explicitly. Overall, the arithmetic used by the system is relatively simple but is nested behind many layers of abstraction, which makes it difficult to analyze without a background in functional programming in Solidity. Comments are scarce, but both unit and invariant tests are thorough. The implementation contains a large number of integer casts and wraps, which increases the complexity. We recommend reviewing the current implementation and reorganizing it to decrease the casting and wrapping. Moving away from the user-defined value types (which are unwrapped, wrapped, and casted on many occasions) would be a good first step to achieve this goal.	**Moderate**
Auditing	The system emits sufficient events to facilitate off-chain monitoring, although minor gaps (TOB-SUPERFLUID-1) and inconsistencies (TOB-SUPERFLUID-2) were identified. Minimal documentation was provided about the implications of each event or what kinds of events could indicate abnormal conditions. Superfluid provided a detailed incident response plan and has an off-chain monitoring system that uses various tools such as Alchemy webhooks, Prometheus, Grafana, and custom monitoring scripts.	**Moderate**

Authentication / Access Controls	The responsibilities of all actors are well defined and tightly controlled. There are few single points of failure because every GDA pool has a different admin, although the GDA and pool logic contracts are owner upgradable. We identified no missing access controls, and restricted functions follow the principle of least privilege.	**Satisfactory**
Complexity Management	The call flows through the different contracts and libraries are sometimes so large that it is hard to follow the entire flow. Each individual function in the protocol is tightly scoped for easy unit testing, but the large number of small functions causes the execution flow to frequently cross file boundaries, which makes it difficult to keep track of. There is a middle ground between very large and very small functions. We recommend trying to achieve the middle ground. Most variables in the `SemanticMoney` library have nondescript names, and the same variable names are used to describe arguments of different types in different contexts. There are numerous upcasts, downcasts, wraps, and unwraps of integer values throughout the implementation, which makes it difficult to follow. Consider rewriting the implementation so that casts and wraps are minimized to the exact places where they are necessary. The decision to use a functional programming style in Solidity makes the implementation more difficult to understand than the typical imperative style. Consider moving away from a functional programming style to decrease the implementation's complexity.	**Weak**
Decentralization	The pool administrators have limited privileges and cannot unilaterally seize user funds. However, the logic governing the GDA and pool contracts is upgradable by a single entity. Given the lack of a time delay on these upgrades, users must trust the Superfluid development team while using the system. We recommend updating the user-facing documentation	**Moderate**

	to describe the current functioning of the governance component, including what actions governance can take. Superfluid has indicated that they have plans to decentralize the governance of the protocol.	
Documentation	Thorough high-level documentation describing the features provided by GDA is provided by both a public documentation website and the wiki associated with the repository. However, lower-level technical specifications are largely absent and inline documentation is sparse. For example, NatSpec comments are not used and many functions are entirely undocumented. This is especially troublesome due to the complex nature of libraries such as `TokenMonad` and `SemanticMoney`.	**Weak**
Low-Level Manipulation	Assembly is used sparingly. The assembly that is present often consists of a single `sload` or `sstore` operation and is justified due to the optimized storage patterns used. Low-level calls are similarly rare and used via well-established libraries only for interacting with proxies and libraries. However, the custom method of tightly packing variables using bitwise operations is used in multiple places in the GDA contract. We identified one flaw (TOB-SUPERFLUID-5) in this mechanism that could pose a problem in a future upgrade. Additionally, we identified an off-by-one error in the low-level checking of reverts due to an out-of-gas error (TOB-SUPERFLUID-6), which could also pose a problem in a future upgrade.	**Moderate**
Testing and Verification	The codebase under review demonstrates a wide range of testing capabilities. Unit tests are thorough, and invariant tests use both Foundry and Echidna fuzz testing frameworks. Static analysis is configured to use Slither, and the automated CI pipeline runs a comprehensive and diverse suite of tests.	**Satisfactory**

Transaction Ordering	The architectural decisions apparent in the GDA's design display an awareness of transaction-timing implications. The system relies on time-based incentives to overcome certain limitations of blockchain platforms, and no timing-based vulnerabilities were uncovered during this review.	**Satisfactory**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Lack of event generation	Auditing and Logging	**Informational**
2	Incorrect event emission in connectPool	Auditing and Logging	**Informational**
3	Lack of two-step process for contract ownership change	Data Validation	**Informational**
4	Error-prone initialization of the SuperfluidUpgradeableBeacon owner	Data Validation	**Informational**
5	Large encoded buffer amount could manipulate preceding field	Data Validation	**Informational**
6	Off-by-one in gas left check	Error Reporting	**Informational**

Detailed Findings

1. Lack of event generation	
Severity: **Informational**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-SUPERFLUID-1
Target: `SuperfluidPool.sol`	

Description

Two critical operations do not emit events. This creates uncertainty among the users interacting with the system.

In figure 1.1, the `operatorSetIndex` function in the `SuperfluidPool` contract does not emit an event when it updates the critical storage variable `_index`. However, having an event emitted to reflect such a change in the critical storage variable may allow other system/off-chain components to detect suspicious behavior in the system.

Events generated during contract execution aid in monitoring, baselining of behavior, and detecting suspicious activity. Without events, users and blockchain-monitoring systems cannot easily detect behavior that falls outside the baseline conditions; malfunctioning contracts and attacks could go undetected.

```
    function operatorSetIndex(PDPoolIndex calldata index) external onlyGDA
returns (bool) {
        _index = _pdPoolIndexToPoolIndexData(index);

        return true;
    }
```

Figure 1.1: The `operatorSetIndex` function in `SuperfluidPool.sol`

The `OperatorConnectMember` function in `gdav1/SuperfluidPool.sol` should also emit an event.

Recommendations

Short term, add events for all functions that change state to aid in better monitoring and alerting.

Long term, ensure all state changing operations are always accompanied by events. In addition, use static analysis tools such as Slither to help prevent such issues in the future.

2. Incorrect event emission in connectPool

Severity: **Informational**	Difficulty: **Medium**
Type: Auditing and Logging	Finding ID: TOB-SUPERFLUID-2
Target: `GeneralDistributionAgreementV1.sol`	

Description

The `connectPool` function may emit an event even when no state changes have been made. This could confuse off-chain monitoring and alerting systems that use events to track the correct functioning of the smart contract.

The `connectPool` function can be used to either connect or disconnect a member to or from a pool. For a connection, a check is performed to ensure the member is not already connected. For a disconnection, a check is performed to ensure the member is currently a member of the pool. When either of these checks fail, the execution continues and no state changes are performed. However, the `PoolConnectionUpdated` event at the end of the function is emitted regardless of whether the check succeeds or fails.

```
    function connectPool(ISuperfluidPool pool, bool doConnect, bytes calldata ctx)
        public
        returns (bytes memory newCtx)
    {
        ISuperfluidToken token = pool.superToken();
        ISuperfluid.Context memory currentContext = AgreementLibrary.authorizeTokenAccess(token, ctx);
        address msgSender = currentContext.msgSender;
        newCtx = ctx;
        if (doConnect) {
            if (!isMemberConnected(token, address(pool), msgSender)) {
assert(SuperfluidPool(address(pool)).operatorConnectMember(msgSender, true, uint32(block.timestamp)));

                uint32 poolSlotID =
                    _findAndFillPoolConnectionsBitmap(token, msgSender, bytes32(uint256(uint160(address(pool)))));

                // malicious token can reenter here
                // external call to untrusted contract
                // what sort of boundary can we trust
                token.createAgreement(
                    _getPoolMemberHash(msgSender, pool),
```

```
                    _encodePoolMemberData(PoolMemberData({ poolID: poolSlotID, pool: address(pool) }))
                );
            }
        } else {
            if (isMemberConnected(token, address(pool), msgSender)) {
                assert(SuperfluidPool(address(pool)).operatorConnectMember(msgSender, false, uint32(block.timestamp)));
                (, PoolMemberData memory poolMemberData) = _getPoolMemberData(token, msgSender, pool);
                token.terminateAgreement(_getPoolMemberHash(msgSender, pool), 1);

                _clearPoolConnectionsBitmap(token, msgSender, poolMemberData.poolID);
            }
        }

        emit PoolConnectionUpdated(token, pool, msgSender, doConnect, currentContext.userData);
    }
```

Figure 2.1: The `connectPool` function in `GeneralDistributionAgreementV1.sol`

Exploit Scenario

Alice never connected her account to the `0xabc` pool but later forgets this. She calls `connectPool` to disconnect her account from `0xabc`. The transaction succeeds and the `PoolConnectionUpdated` event is emitted. Off-chain monitors tracking this event add Alice's account to the list of accounts that have been disconnected from the `0xabc` pool and show this information in their web applications.

Recommendations

Short term, do one of the following:

- Rearrange the logic so that the `PoolConnectionUpdated` event is not emitted when there have been no pool membership state changes.
- Revert the transaction when trying to connect someone who is already a member of the pool or when trying to disconnect someone who is not a member of the pool.

Long term, emit events only at the right occasions and with accurate data so that less time needs to be spent setting up off-chain monitoring. However, off-chain monitoring can help track the correct functioning of the smart contracts and can also be used to review previous state changes that were made in the smart contract, which could be useful in case of a security incident.

3. Lack of two-step process for contract ownership change

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SUPERFLUID-3

Target: `packages/ethereum-contracts/contracts/agreements/gdav1/SuperfluidPoolDeployerLibrary.sol`

Description

The `SuperfluidUpgradableBeacon` contract inherits OpenZeppelin's `Ownable` contract via its `UpgradeableBeacon` dependency. The owner of such contracts can be changed through a call to the `transferOwnership` function, which immediately sets the contract's new owner. Making such a critical change in a single step is error-prone and can lead to mistakes that are difficult to recover from.

```
contract SuperfluidUpgradeableBeacon is UpgradeableBeacon {
```

Figure 3.1: The `SuperfluidUpgradeableBeacon` contract inherits OpenZeppelin's `UpgradeableBeacon` contract (`SuperfluidUpgradeableBeacon.sol`)

```
contract UpgradeableBeacon is IBeacon, Ownable {
```

Figure 3.2: The `UpgradeableBeacon` contract inherits OpenZeppelin's `Ownable` contract in `UpgradeableBeacon.sol`

```
function _transferOwnership(address newOwner) internal virtual {
    address oldOwner = _owner;
    _owner = newOwner;
    emit OwnershipTransferred(oldOwner, newOwner);
}
```

Figure 3.3: The internal helper of the `transferOwnership` function transfers ownership in a single step in OpenZeppelin's `Ownable.sol`.

Exploit Scenario

Alice, a Superfluid administrator, attempts to migrate ownership of the `SuperfluidPool` proxy contract to a different account. She mistakenly transfers ownership to an incorrect, invalid address. As a result, the ownership role is permanently lost. The logic contract of this proxy can now no longer be updated, all existing pools must migrate to use a newly deployed proxy contract.

Recommendations

Short term, override the `transferOwnership` function in the `SuperfluidUpgradableBeacon` contract with one that performs the first step of a two-step ownership process, and add a new `acceptOwnership` function that performs the second step. This will ensure that ownership cannot be transferred to an invalid address.

Long term, do not trust deployment or upgrade scripts. They can contain bugs, and the humans executing them can do so incorrectly or unsafely. Any security-critical safeguards that can be enforced on-chain should be enforced.

4. Error-prone initialization of the SuperfluidUpgradeableBeacon owner

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SUPERFLUID-4
Target: `packages/ethereum-contracts/contracts/agreements/gdav1/SuperfluidPoolDeployerLibrary.sol`	

Description

OpenZeppelin's `Ownable` contract sets the initial owner to the `msg.sender` address during deployment, as shown in figure 4.1. A private key accessed from the environment in `hardhat.config.js` (figure 4.2) suggests that the deployer private key will be available via the process environment, which puts the initial owner's private key and user funds at risk.

```
constructor() {
    _transferOwnership(_msgSender());
}
```

Figure 4.1: The `constructor` in OpenZeppelin's `Ownable.sol`

Keeping secrets in environment variables is a well-known antipattern. Environment variables are commonly captured by all manner of debugging and logging information, can be accessed from `procfs` tool, and are passed down to all child processes.

```
function createNetworkConfig(
   network: keyof typeof chainIds
): NetworkUserConfig {
   return {
       accounts:
           process.env.PRIVATE_KEY !== undefined
               ? [process.env.PRIVATE_KEY]
               : [],
       chainId: chainIds[network],
   };
}
```

Figure 4.2: The private key is read from the environment in `hardhat.config.ts`.

Exploit Scenario

Alice, a Superfluid developer, deploys the GDA feature including the `SuperfluidUpgradeableBeacon` contract for Superfluid pools. Unbeknownst to her, one of the deployment script's dependencies has a malicious dependency and it exports her private key to an attacker. Users begin using Superfluid pools to distribute their funds, and once enough value is being managed, the attacker upgrades all pools to forward all distributions to their wallet instead.

Recommendations

Short term, add a new argument to the constructor of the `SuperfluidUpgradeableBeacon` contract that specifies the initial owner. This account should be a multisignature or an address managed by a hardware wallet, and it should be set as the owner of the `SuperfluidUpgradeableBeacon` contract before the deployment transaction completes.

Long term, do not trust deployment scripts. Since it is not feasible to rely on the security of the software supply chain, smart contracts should be designed so that the deployer address does not have any special privileges.

5. Large encoded buffer amount could manipulate preceding field

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SUPERFLUID-5
Target: `packages/ethereum-contracts/contracts/agreements/gdav1/GeneralDistributionAgreementV1.sol`	

Description

Forgetting to mask the first 160 bits of the `totalBuffer` field when packing it tightly into a `bytes32` value as an `int96` allows the preceding field to be manipulated. Currently this is not exploitable, but since the contracts are upgradable, a future update to the implementation could make it exploitable.

The `UniversalIndexData` struct contains five fields that are tightly packed into two `bytes32` values. Figure 5.1 shows the struct definition.

```
struct UniversalIndexData {
   int96 flowRate;
   uint32 settledAt;
   uint256 totalBuffer;
   bool isPool;
   int256 settledValue;
}
```

Figure 5.1: The `UniversalIndexData` struct in `IGeneralDistributionAgreementV1.sol`

The above struct is tightly packed using the code in figure 5.2. Since `totalBuffer` is a `uint256` and it is shifted to the left 32 bits, the last 224 bits of `totalBuffer` will be written into the `bytes32` value using a bitwise OR operation. This could potentially alter the `flowRate` and `settledAt` values. Due to the use of a bitwise OR operation, altering the values does not allow total control over the values, but it does allow increasing the `flowRate` and `settledAt` values.

```
function _encodeUniversalIndexData(UniversalIndexData memory uIndexData)
    internal
    pure
    returns (bytes32[] memory data)
{
    data = new bytes32[](2);
    data[0] = bytes32(
        (uint256(int256(uIndexData.flowRate)) << 160) |
        (uint256(uIndexData.settledAt) << 128) |
        (uint256(uIndexData.totalBuffer) << 32) |
```

```
        (uIndexData.isPool ? 1 : 0)
    );
    data[1] = bytes32(uint256(uIndexData.settledValue));
}
```

Figure 5.2: Reformatted version of the `_encodeUniversalIndexData` function in `GeneralDistributionAgreementV1.sol`

The `totalBuffer` amount is not directly user-controllable but depends on a possible previous buffer amount, the user-controllable `int96 requestedFowRate` variable and the governance-controllable `uint32 liquidationPeriod` variable (which is set to 14400 in the provided tests). Figure 5.3 shows how these values are used to determine the new `totalBuffer` amount (the `Value` type is an `int256`).

```
    Value newBufferAmount = newFlowRate.mul(Time.wrap(uint32(liquidationPeriod)));

    if (Value.unwrap(newBufferAmount).toUint256() < minimumDeposit && FlowRate.unwrap(newFlowRate) > 0) {
        newBufferAmount = Value.wrap(minimumDeposit.toInt256());
    }

    Value bufferDelta = newBufferAmount - Value.wrap(uint256(flowDistributionData.buffer).toInt256());

    {
        bytes32[] memory data = _encodeFlowDistributionData(
            FlowDistributionData({
                lastUpdated: uint32(block.timestamp),
                flowRate: int256(FlowRate.unwrap(newFlowRate)).toInt96(),
                buffer: uint256(Value.unwrap(newBufferAmount)) // upcast to uint256 is safe
            })
        );

        ISuperfluidToken(token).updateAgreementData(flowHash, data);
    }

    UniversalIndexData memory universalIndexData = _getUIndexData(eff, from);
    universalIndexData.totalBuffer =
    // new buffer
    (universalIndexData.totalBuffer.toInt256() + Value.unwrap(bufferDelta)).toUint256();
```

Figure 5.3: Excerpt from the `_adjustBuffer` function in `GeneralDistributionAgreementV1.sol`

As a result of the restrictions imposed by the `requestedFlowRate` and `liquidationPeriod` variables' types, the maximum new totalBuffer amount is `type(int96).max * 14400 = 570442770102703230673516442404800` (as bytes: `0x1c1fffffffffffffffffffffc7c0`), which is above the maximum value of an `int96`.

Therefore, the bytes `0x1c1f` are used in a bitwise OR operation with the preceding `settledAt` field's value (thereby increasing it).

Currently this issue is not exploitable because the increased `settledAt` value (which was originally the `block.timestamp` variable) being subtracted from the current timestamp will lead to an underflow and revert.

We used one of the two `_encodeUniversalIndexData` functions in the preceding explanation. However, the ability to alter the preceding field also applies to the other `_encodeUniversalIndexData` function and the `_encodeFlowDistributionData` function (where the preceding `flowRate` value can be manipulated).

Exploit Scenario

A future upgrade of the implementation swaps the `isPool` and `settledAt` fields in the encoding of the `UniversalIndexData` struct. Using a carefully chosen `requestedFlowRate` value, an attacker sets the `isPool` field on their encoded `UniversalIndexData` struct from zero to one.

Recommendations

Short term, use the safe `.toInt96()` function to explicitly downcast the `uint256` buffer value in all the encoding functions before using a bitwise OR operation to move it into the `bytes32` value. Also, consider reformatting the encoding functions as shown in figure 5.2. This will improve the readability and may help identify similar bugs during development.

Long term, always safely downcast and correctly mask values when manually tightly packing values. Failing to do so could result in unexpected behavior, with the potential of rendering a contract inoperable, manipulating internal accounting, or stealing funds.

6. Off-by-one in gas left check

Severity: **Informational**	Difficulty: **High**
Type: Error Reporting	Finding ID: TOB-SUPERFLUID-6
Target: `packages/ethereum-contracts/contracts/libs/SafeGasLibrary.sol`	

Description

There is an off-by-one error in the check to see if a call reverted due to insufficient gas. If a call reverted because of another reason, it could be interpreted as having reverted due to out-of-gas. In the current implementation, this does not pose a problem; however, in a future upgrade, it could become a problem, albeit one of very low severity.

The EVM by default will pass along 63/64 of the total gas available to any callee in order to prevent a Call Depth Attack.

The Superfluid protocol makes various calls to contracts that may revert. To ensure the transaction reverts only if it runs out of gas, a try-catch statement is used that will, in case of a revert, check whether it was due to insufficient gas or some other reason. Only if it runs out of gas does the transaction revert. These calls could therefore be seen as optional.

To check that a call reverted due to insufficient gas, the amount of gas before the call is saved. Then the call is made, and if it reverts, a check is performed to see whether the gas after the call is less than 1/63 of the gas amount before the call (shown in figure 6.1). This is an off-by-one error since the correct value to check against would be 1/64.

```
    function _isOutOfGas(uint256 gasLeftBefore) internal view returns (bool) {
        return gasleft() <= gasLeftBefore / 63;
    }

    /// @dev A function used in the catch block to handle true out of gas errors
    /// @param gasLeftBefore the gas left before the try/catch block
    function _revertWhenOutOfGas(uint256 gasLeftBefore) internal view {
// If the function actually runs out of gas, not just hitting the safety gas limit, we revert the whole transaction.
// This solves an issue where the gas estimaton didn't provide enough gas by default for the function to succeed.
// See https://medium.com/@wighawag/ethereum-the-concept-of-gas-and-its-dangers-28d0eb809bb2
```

```
        if (_isOutOfGas(gasLeftBefore)) {
            revert OUT_OF_GAS();
        }
    }
```

Figure 6.1: The out-of-gas related functions in `SafeGasLibrary.sol`

In the current implementation, the called contracts cannot trigger this issue, and the Superfluid protocol has a dedicated test to ensure it cannot happen.

Exploit Scenario

An upgrade to the implementation of the `ConstantOutflowNft.onCreate` function adds a check at the end of the function with a revert error called `SOME_ERROR`. Alice sets up a distribution, and this new check in `onCreate` reverts. Due to the off-by-one error, the revert cause is interpreted as insufficient gas and the `OUT_OF_GAS` error is returned, when actually it was the `SOME_ERROR` error that caused the revert.

Recommendations

Short term, replace `gasLeftBefore / 63` with `gasLeftBefore / 64`.

Long term, be aware of the inner workings and correctly implement any feature that uses a low-level EVM feature.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality

The following areas for improvement are not associated with specific vulnerabilities. However, addressing them would enhance code readability and may prevent the introduction of vulnerabilities in the future.

- **Remove the masking of the `flowRate` and `lastUpdated` values.** Since these variables are at the front of the packed `bytes32` value, there is no need to mask the fronting bits.

```
universalIndexData.flowRate = int96(int256(a >> 160) & int256(uint256(type(uint96).max)));
```

Figure C.1: The masking of the `flowRate` value in GeneralDistributionAgreementV1.sol

```
flowDistributionData.lastUpdated = uint32((data >> 192) & uint256(type(uint32).max));
```

Figure C.2: The masking of the `lastUpdated` value in GeneralDistributionAgreementV1.sol

- **Remove the & 1 part.** Simply checking against == 1 will give the same result.

```
function _isPool(ISuperfluidToken token, address account) internal view returns (bool exists) {
    // @note see createPool, we retrieve the isPool bit from
    // UniversalIndex for this pool to determine whether the account
    // is a pool
    exists = (
        (uint256(token.getAgreementStateSlot(address(this), account, _UNIVERSAL_INDEX_STATE_SLOT_ID, 1)[0]) << 224)
            >> 224
    ) & 1 == 1;
}
```

Figure C.3: The `_isPool` function in GeneralDistributionAgreementV1.sol

D. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

From December 5 to December 7, 2023, Trail of Bits reviewed the fixes and mitigations implemented by the Superfluid Finance team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the six issues described in this report, Superfluid Finance has resolved three and has not resolved the remaining three issues. For additional information, refer to the Detailed Fix Review Results section that follows.

ID	Title	Severity	Status
1	Lack of event generation	Informational	Unresolved
2	Incorrect event emission in connectPool	Informational	Resolved
3	Lack of two-step process for contract ownership change	Informational	Unresolved
4	Error-prone initialization of the SuperfluidUpgradeableBeacon owner	Informational	Unresolved
5	Large encoded buffer amount could manipulate preceding field	Informational	Resolved
6	Off-by-one in gas left check	Informational	Resolved

Detailed Fix Review Results

TOB-SUPERFLUID-1: Lack of event generation
Unresolved. The Superfluid Finance team provided the following context for this finding's fix status:

> *We previously had events for this but explicitly decided against this because we felt the call/event was too low level and didn't need to be exposed.*

TOB-SUPERFLUID-2: Incorrect event emission in connectPool
Resolved. The `connectPool` method of the `GeneralDistributionAgreementV1` contract emits an event only when the connection status changes.

TOB-SUPERFLUID-3: Lack of two-step process for contract ownership change
Unresolved. The Superfluid Finance team provided the following context for this finding's fix status:

> *We didn't want to introduce a new pattern which we have not applied anywhere.*

TOB-SUPERFLUID-4: Error-prone initialization of the SuperfluidUpgradeableBeacon owner
Unresolved. The Superfluid Finance team provided the following context for this finding's fix status:

> *We `init` the ownership to `msg.sender` and then immediately transfer ownership to the Superfluid host contract.*

TOB-SUPERFLUID-5: Large encoded buffer amount could manipulate preceding field
Resolved. The buffer is now explicitly cast to a `uint96` before being shifted. In the case of an overflow, the method will now revert instead of resulting in undefined behavior.

TOB-SUPERFLUID-6: Off-by-one in gas left check
Resolved. The `_isOutOfGas` method in the `SafeGasLibrary` contract now compares the total gas left to `gasLeftBefore` divided by 64, aligning this check with the underlying EVM behavior.

E. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.